Multi-threading

Multitasking vs Multithreading

Multi-tasking

- Executing multiple tasks(processes) concurrently or paralelly.
- Here each task/process represents an application on computer. Each task has a process ID.
- Performs on Operating System level.
  Example : Running a web browser, text-editor, music player at the same time on a computer.

Multi-threading

- Executing multiple threads on a single task or a single program concurrently or paralelly.
- Here task or program reperesents a block of java code or web request
- Performs on Program/Application level.
  Example : A web server handling multiple requests simultaneously using separate threads for each request.

Note : Thread is a smallest unit of proccessing. OS is not going to allocate any kind of resource or Memory or CPU specifically or individually for a thread. Threads are sharing the resources got allocated to the process.

Benefits of Multithreading

1. Responsiveness : Multithreading improves the responsiveness of applications, particularly in user interfaces.For example, a Word application takes user input and spell checks in background without freezing.
2. Asynchronous Processing : Multithreading allows for asynchronous execution, which is useful for tasks such as I/O operations, network communication, and file handling, enabling the main thread to continue processing other tasks.
3. Paralellism : Multithreading enables true parallelism, where multiple threads executes task paralelly which leads to faster task completion.

4. PErformance Improved : Multithreading can significantly enhance the performance of applications by utilizing multiple CPU cores. Tasks can be executed in parallel, reducing the overall time taken to complete a set of operations.
5. Resource Sharing : Threads within the same process share the same memory space, allowing them to easily share data and resources without complex inter-process communication.
6. Scalability : As the workload increases, multithreaded applications can scale by adding more threads to handle additional tasks. This is particularly beneficial in cloud environments where resources can be scaled up dynamically based on demand.

Java Thread Model

1. Extends Thread class (java.lang.Thread)
2. Implements Runnable interface (java.lang.Runnable) - Functional Interface - run()

Extending Thread class

```
class ExtendingThreadClassDemo{
    public static void main(String[] args){
        //Intializing Thread
        MyThread t1 = new MyThread();

        // Starting Thread
        t1.start();
    }

}
// Defining Thread
class MyThread extends Thread{

    //Thread job
    @Override
    public void run(){
        for(int i=0;i<5;i++>){
            System.out.println(i);
        }
    }
}
```

Implementing Runnable Interface

```
class ImplementingRunnableInterfaceDemo{
    public static void main(String[] args){
        //Intializing Thread
        MyRunnable task = new MyRunnable();
        Thread t1 = new Thread(task);
        // Starting Thread
        t1.start();
    }

}
// Defining Thread
class MyRunnable implements Runnable{

    //Thread job
    @Override
    public void run(){
        for(int i=0;i<5;i++>){
            System.out.println(i);
        }
    }
}
```

Using Anonymous class

```
class ImplementingRunnableInterfaceDemo{
    public static void main(String[] args){
        //Intializing Thread (Make use of Anonymous class)
        Thread t1 = new Thread(new Runnable(){
            //Thread job
            public void run(){
                    System.out.println("Thread generated using Anonymous class");
                    System.out.println(Thread.currentThread().getName());
            }
        });
        // Starting Thread
        t1.start();
    }

}
```

Using Java 8

```
class ImplementingRunnableInterfaceDemo{
    public static void main(String[] args){
        //Intializing Thread (Make use of lambda expression)
        Thread t1 = new Thread(() -> {
            //Thread job
            System.out.println("Thread generated using Java 8");
            System.out.println(Thread.currentThread().getName());
        });
        // Starting Thread
        t1.start();
    }

}
```

start() vs run()

start()

- Register thread with the thread schedular
- Perform all other mandatory task()

- invoke run()

run()

- Thread job

Lifecycle of Thread

Phase 1 - *Newborn state -* MyThread t1 = new MyThread();
Phase 2 - *Runnable state -* t1.start();
Phase 3 - *Running state -* Thread schedular will allocate resources to thread
Phase 4 - *Dead state -* Thread job is completed

Note : Whenever we are trying to call start() on same thread multiple time then we may get IllegalThreadStateException.

Thread class has 8 Constructors

```
1. public Thread()
2. public Thread(Runnable target)
3. public Thread(Runnable target, String name)
4. public Thread(String name)
5. public Thread(ThreadGroup group, Runnable target)
6. public Thread(ThreadGroup group, Runnable target, String name)
7. public Thread(ThreadGroup group, String name)
8. public Thread(ThreadGroup group, Runnable target, String name, long stackSize)
```